Unofficial X-Chat Scripting Documentation

Created: February 15th, 1999.
Last Modified: July 30th, 1999.

Documentation Author: Dagmar d'Surreal
Updated By: Peter Zelezny

Perl

RFC 1459
nice list of server numerics changes for Hybrid 6

Unofficial X-Chat Programming Documentation

Version 1.4
By Dagmar d'Surreal
March 26th, 1998
Update By Peter Zelezny
July 30th, 1999

Current version of X-Chat: 1.1.7
Official Homepage: X-Chat Homepage
Author: zED

Necessary Packages:
Glib and GTK+ Stable Tree
(version 1.2.3 recommended)

Optional Packages:
Gnome

```
----------from perl.c----------
XS(XS_IRC_register);
XS(XS_IRC_add_message_handler);
XS(XS_IRC_add_command_handler);
XS(XS_IRC_add_timeout_handler);
XS(XS_IRC_add_print_handler);
XS(XS_IRC_print);
XS(XS_IRC_send_raw);
XS(XS_IRC_command);
XS(XS_IRC_command_with_server);
XS(XS_IRC_channel_list);
XS(XS_IRC_server_list);
XS(XS_IRC_user_list);
XS(XS_IRC_dcc_list);
XS(XS_IRC_get_info);
------------------------------
newXS("IRC::register", XS_IRC_regist
newXS("IRC::add_message_handler", XS
newXS("IRC::add_command_handler", XS
newXS("IRC::add_timeout_handler", XS
newXS("IRC::add_print_handler", XS_I
newXS("IRC::print", XS_IRC_print, "I
newXS("IRC::send_raw", XS_IRC_send_r
newXS("IRC::command", XS_IRC_commanc
newXS("IRC::command_with_server", XS
newXS("IRC::channel_list", XS_IRC_ch
newXS("IRC::server_list", XS_IRC_ser
newXS("IRC::user_list", XS_IRC_user_
newXS("IRC::dcc_list", XS_IRC_dcc_li
newXS("IRC::get_info", XS_IRC_get_ir
------------------------------
```

Standard Disclaimer

This documentation is provided on an "as-is" basis and comes with no warran
or usefulness, either expressed or implied. It is subject to change without an
may contain omissions or errors which could cause your genitalia to shrivel a
spontaneously combust. If you have any further questions,
please feel free to seek professional help.

Perl

X-Chat uses perl for scripting, and pretty much only perl. There's support for writing modules, now, but it's not quite as simple to stick a module together in C and make it stable compared to the development time of perl code. For those of you who are used to ircII's scripting language, or using eggdrop with tcl, the only thing you will really miss is that the regexps are a lot different, but they're more powerful in perl. For those of you saddled with weak languages for too long, regexps means REGular EXPressions, or "pattern matching wildcard usage" in layman's terms.

Handlers

There are [currently] three basic ways to make things call the subroutines you write for X-Chat and they are message handlers, command handlers, and timeout handlers. Message handlers trigger on messages sent from the IRC server to your client, command handlers are triggered by / commands typed in by the user at the keyboard, and timeout handlers are actually triggered by gtk+.

Exit Codes

These are very important. Every time you set up a handler, it takes precedent over the built-in functions and commands of X-Chat. That is, whatever thing which triggered your subroutine will go to your code before it goes to X-Chat to be dealt with. In this way you can replace almost every built-in function that the client has with your own routines. The thing to remember is that if your code exits by hitting the end of your subroutine, or by a plain 'return' statement, processing of the event will go on to whatever other things have set up hooks for the event, and then (provided nothing else exits with a return value of 1) to X-Chat itself. There is only one problem with this, (which is solved by the brokering handler that I'll explain that later) and that is that you cannot really control what order the custom routines get called. Normally they will execute in order of which ones were installed first, but a single script has no real way of knowing this. Beware.

@_

If you've never heard of @_ before, then you've obviously not coded in perl. When a message handler triggers, the raw line from the IRC server is passed to the subroutine you specify in @_. When a command handler is triggered, only the arguments are passed to the routine through @_ and they are not broken into a list, but left as one long string. You'll have to parse those yourself with split. (I advise using s/\s+/ /g to collapse the blank space to single space first.) When a timer handler is triggered, I *think* absolutely nothing is passed in @_, but it's not like anything terrifically important could be passed along anyway. Be especially careful when setting up message handlers for mode changes, since the modes are not broken up into individual events like they are with eggdrop. The upside of this is that X-Chat has no mode hooks of it's own, so you don't have to worry about it too much. (This is not the case with the brokering handler, however.)

Timeout Handlers

Timeout handlers are worthy of special notice to people who are used to coding for ircII/EPIC/BitchX/etc. They do *not* function the same way in X-Chat. Timeout handlers are triggered only once. If you put in a timeout handler with an interval of 1000, then your subroutine will be triggered only once, in 1 seconds time. Be careful not to use an exceptionally low timer, since this will drive the CPU load up unnecessarily.

Contexts

There are some really nice things about coding for X-Chat, and the biggest one is that it's fairly good about determining the proper context for things. If a server sends something that triggers a message handler, then you can be sure that unless you specify otherwise, that your IRC::print or IRC::command function call will go back to that server and that server alone. If you really really need to know what the current context is, use the IRC::get_info function as detailed below.

And now the functions themselves...

IRC::register(*scriptname*, *version*, *shutdownroutine*, *ununsed***);**

This is the first function your script should call, example:

IRC::register ("my script", "1.0", "", "");

The "*shutdownroutine*" arg is a function that will be called when X-Chat shuts down, so you get a chance to save config files etc. You can omit this arg, it is optional. The "*unused*" arg is reserved for future use, for now just provide "". This function also returns X-Chat's version number.

IRC::add_message_handler(*message*, *subroutine_name***);**

This function allows you to set up hooks to subroutines so that when a particular message arrives from the IRC server that you are connected to, it can be passed to a subroutine to be dealt with appropriately. The message argument is essentially the second solid token from the raw line sent by the IRC server, and X-Chat doesn't know that some numeric messages have associated text messages, so for now set up a handler for both if you want to be sure odd servers don't screw up your expectations. (Read: fear IRCNet.) The entire line sent by the IRC server will be passed to your subroutine in @_. For the completely uninitiated, *message*s are things like 'PRIVMSG', 'NOTICE', '372', etc.

IRC::add_command_handler(*command*, *subroutine_name***);**

This function allows you to set up hooks for actual commands that the user can type into the text window. The arguments are passed to the subroutine via @_, and arrive as a single string. @_ will be null if no arguments are supplied. It's recommended that you be sure and collapse the excess whitespace with s/\s+/ /g before attempting to chop the line up with split. As mentioned earlier, exiting with an undefined return value will allow the command to be parsed by other handlers, while using a return value of 1 will signal the program that no further parsing needs to be done with this command.

IRC::add_timeout_handler(*interval*, *subroutine_name***);**

This function is one to pay careful attention to when you use it. This is a one-shot timer, it must be reinstalled if you want it to be called regularly. The interval is measured in milliseconds, so don't use a particularly small value unless you wish to drive the CPU load through the roof. 1000ms = 1 second. No values will be passed to the routine via @_ and return values don't affect anything either.

IRC::add_print_handler(*message*, *subroutine_name***);**

This function allows you to catch the system messages (those who generally start by three stars) and to execute a function each time an event appear. The events are those you can see in "Settings->Edit Events Texts". message is the name of the event (you can find it in the Edit Events box, "Events" column) , subroutine_name is the name of the function that will get messages. Be carrful: all the arguments are sent to function in $_[0] separated by spaces.

IRC::print(*text***);**

This is a very simple routine. All it does is put the contents of the text string to the current window. The current window will be whichever window a command was typed into when called from a command handler, or in whichever window the message command is appropriate to if it is called from within a message handler. As with any perl program, newlines are not assumed, so don't forget to end the line with \n if you don't want things to look screwey.

IRC::send_raw(*text***);**

This routine is very useful in that it allows you to send a string directly to the IRC server you are connected to. It is assumed that the server will be the one you first connected to if there is no clear context for the command, otherwise it will go to whatever server triggered the message handler or command handler window. You must specify newlines here always or you can be guaranteed that strange things will happen. The text message you specify should be a proper RAW IRC message, so don't play with it if you don't know how to do these. Additionally, while newlines are also not assumed here as with the IRC::print function, the RFC specifies that newlines are a CR+LF pair, even if most servers will accept a mere newline. It's best to play it safe and use \r\n instead of just \n.

IRC::command(*text***);**

This routine allows you to execute commands in the current context. The text string containing the command will be parsed by everything that would normally parse a command, including your own command handlers, so be careful. Newlines are assumed, thankfully.

IRC::command_with_server(*text*, *servername***);**

This routine allows you to specify the context of the server for which the command will be executed. It's not particularly useful unless you're managing a connection manually, yet the command still exists for it's usefulness in doing things like managing a bnc connection, etc. Newlines are assumed here as well.

IRC::channel_list( );

This command returns a flat list which contains the current channel, server, and nickname for all channels the client is currently in. You'll have to break the list up into groups of three yourself. No arguments are necessary, or used [currently].

IRC::server_list( );

This command returns a flat list of servers you are connected to.

IRC::user_list(*channel*, *server*);

Works very much like the dcc_list command below, except that is returns information about the users on the channel provided as first argument. The second argument is the server and is optional. An extract from perl.c is:

```
XST_mPV(i, user->user); i++;
if(user->hostname)
   XST_mPV(i, user->hostname);
else
   XST_mPV(i, "FETCHING");
i++;
XST_mIV(i, user->op); i++;
XST_mIV(i, user->voice); i++;
XST_mPV(i, ":"); i++;
```

Each user entry is seperated by the ":". If the hostname is not known, the string "FETCHING" will be returned.

IRC::dcc_list( );

This command does essentially the same thing as channel_list, giving you the details of each DCC connection currently in progress. I have no idea exactly what is returned because I haven't had a chance to poke at this one much, but suffice it to say that it's a flat list, and the first time you play with it the meaning of the returned values should be pretty obvious. (Send me email if you decide to dig deeply into this command, thanks!)

Here's a fragment from the perl.c file in the source code which may shed some light as to the nature of the returned values:

```
    if(dcc->nick[0])
     {
       XST_mPV(i, dcc->nick);
       i++;
       XST_mPV(i, dcc->file);
       i++;
       XST_mIV(i, dcc->type);
       i++;
       XST_mIV(i, dcc->stat);
       i++;
       XST_mIV(i, dcc->cps);
       i++;
       XST_mIV(i, dcc->size);
       i++;
       XST_mIV(i, dcc->resumable);
       i++;
       XST_mIV(i, dcc->addr);
       i++;
       XST_mPV(i, dcc->destfile);
       i++;
     }
```

Hopefully this will be enough for people to figure out what's going on, but just in case, check out the dp_misc.pl file which will make this thing output a much more understandable function through an OO wrapper.

IRC::get_info(*integer*);

This function returns a bit of selected information depending on what the value of the integer is. Here's a list of the currently supported values:

0 - The version of X-Chat you're running ("1.1.7" currently).
1 - The nickname you are using in the current server context, or the default nickname if you're not connected to a server.
2 - The current channel context, if you're in a channel, or null if you're not or there is no channel context.
3 - The name of the IRC server you're connected to currently, or null if you're not connected to a server.
Anything else - Zilch.